Turning Compositionality into Composability

Björn Andersson
Carnegie Mellon University

ABSTRACT

Compositional theories and technologies facilitate the decomposition of a complex system into components, as well as their integration via interfaces. Component interfaces hide the internal details of the components, thereby reducing integration complexity. A system is said to be composable if the properties established and validated for components in isolation hold once the components are integrated to form the system. This brings us the question: "Is compositionality related to composability?" This paper answers this question in the affirmative; it considers a previously known interface for compositionality and shows that it can be used for composability. It also presents a run-time policing mechanism for this interface.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; G.4 [**Mathematical Software**]: Algorithm design and analysis

General Terms

Algorithms, Performance, Theory

Keywords

Real-time, Composability, Compositionality

1. INTRODUCTION

Consider a taskset τ with each task in this taskset generating a sequence of jobs with each job characterized by an arrival time, an execution time, and a deadline. Assume that there are parameters (e.g., deadline, minimum inter-arrival time, execution time) describing how a task can generate this sequence of jobs. Also, assume that jobs are scheduled on a single processor with preemptive Earliest-Deadline-First (EDF). In addition, assume that jobs do not self-suspend and assume jobs do not share other resources (e.g., critical sections) and assume there is no overhead associated with context switching.

Let the demand-bound function (hereafter referred to by "dbf") of a job for a time interval, I, be equal to its execution time if this job arrives no earlier than the beginning of I and its deadline is no later than the end of I; otherwise the dbf of the job is equal to zero. The dbf of a jobset, for a time interval, I, is the sum, over all jobs in the jobset, of the dbf of the job for I. Let L represent an interval length. The dbf of τ_i for intervals of length L, denoted $\mathrm{dbf}(\tau_i, L)$, is the supremum of the dbf of τ_i taken over all intervals I of length L. The computation of $\mathrm{dbf}(\tau_i, L)$ depends on the types of jobs that τ_i can generate; if τ_i is a sporadic task with minimum inter-arrival time T_i and relative deadline D_i and each of its jobs have execution time C_i, then it is known [5] that $\mathrm{dbf}(\tau_i, L)$ can be computed as $\mathrm{dbf}(\tau_i, L) = \max(\lfloor \frac{L-D_i}{T_i} \rfloor + 1, 0) \times C_i$.

We say that a taskset is *schedulable* if for each jobset that it can generate, for each schedule that EDF can generate for this jobset, it holds that all jobs meet deadlines (Note that because we assume EDF with arbitrary tie-breaking, there may be more than one valid schedule for a jobset scheduled by EDF.) It is known [5] that if for all $L > 0$,

$$\sum_{\tau_i \in \tau} \mathrm{dbf}(\tau_i, L) \leq L, \quad (1)$$

then the taskset is schedulable. The result above (in Eq. 1) is well-known and allows software practitioners to efficiently verify, before run-time, that all timing requirements will be met at run-time. This result works assuming that the entire taskset is known to a single person (or schedulability analysis tool) and that the system does not undergo design changes.

Unfortunately, many systems are designed by many individuals, often spanning multiple organizations with little opportunity to interact. Sometimes one individual cannot change software written by another individual, even if they work on the same large software system. Also, there are requests to change the system in small and large ways ranging from fixing defects to upgrading to new hardware or sensors and adding new functionality. Therefore, it is desirable to perform schedulability analysis in a modular fashion. One way to do so is to describe (and potentially also design) the system as a set of components with each component comprising one or multiple tasks, and let each component offer an interface which describes the resource consumption. The research literature offers many different types of such interfaces.

We now assume that the system is composed of K components, indexed by $1 \leq k \leq K$. Let τ^k represent all the tasks

associated with component k, $1 \leq k \leq K$. We define the component-k dbf for intervals of length L by

$$\text{dbf}(\tau^k, L) = \sum_{\tau_i \in \tau^k} \text{dbf}(\tau_i, L). \quad (2)$$

Combining Eq 1 and Eq 2 yields that: If for all positive L it holds that:

$$\sum_{k \in \{1..K\}} \text{dbf}(\tau^k, L) \leq L \quad (3)$$

then the taskset is schedulable.

The schedulability test expressed by Eq 3 does not require detailed knowledge of the individual tasks and their dbfs in each component. Rather, it only requires knowledge of the dbf associated with each component. If there are changes to any task in any component, they will not impact the schedulability test as long as they do not change the dbf for that component.

Note that in this way, the system integrator does not need to know the internals of a component and can still perform schedulability testing of the system as a whole. Also, note that in this way, the person developing a component k is free to change the software of component k as long as $\text{dbf}(\tau^k, L)$ is not affected.

One can issue three critiques against this method. First, this method assumes that the software is already written (since it requires that one can compute the dbf from the tasks). Second, this method exposes lots of details of a component. This can be problematic because other software (e.g., other software components of the software system or software tools that perform analysis on the software system) may depend on this information that gets exposed and this creates dependencies. Third, this method provides compositionality but not composability.

The first critique is simple to address. A prime contractor can simply state a dbf as an interface for each component and then tell each sub contractor that (s)he must develop software so that the dbf of his/her component is at most as specified by the interface. If not, there is a need for renegotiation between the subcontractor and the prime contractor. The second critique can be addressed by overapproximating the dbf of a component. Indeed, recent studies [3, 4] have provided methods that do this and prove that such methods result in bounded performance loss. And they have proven bounds on how much information they expose. The third critique will be discussed in this paper.

2. A FORMULATION OF COMPOSABILITY

Note that in Eq. 3, the left-hand side is a summation of terms. Choosing one value of $k \in \{1..K\}$ and rearranging the terms in the left-hand side yields:

$$\text{dbf}(\tau^k, L) + \sum_{k' \in \{1..K\} \setminus \{k\}} \text{dbf}(\tau^{k'}, L) \leq L \quad (4)$$

Rewriting yields:

$$\text{dbf}(\tau^k, L) \leq (L - \sum_{k' \in \{1..K\} \setminus \{k\}} \text{dbf}(\tau^{k'}, L)) \quad (5)$$

Hence, if

$$\forall k \in \{1..K\}, (\forall L > 0, \text{dbf}(\tau^k, L) \leq (L - \sum_{k' \in \{1..K\} \setminus \{k\}} \text{dbf}(\tau^{k'}, L))) \quad (6)$$

then the system is schedulable.

In practice, we are often interested in using not the exact demand-bound function but rather an upper bound on the demand-bound function. For each component k, we let $\text{dbfUB}(\tau^k, L)$ be a function such that for all non-negative t it holds that $\text{dbf}(\tau^k, L) \leq \text{dbfUB}(\tau^k, L)$. With this definition of $\text{dbfUB}(\tau^k, L)$, it is easy to see that the following is true: If

$$\forall k \in \{1..K\}, (\forall L > 0, \text{dbfUB}(\tau^k, L) \leq (L - \sum_{k' \in \{1..K\} \setminus \{k\}} \text{dbfUB}(\tau^{k'}, L)))$$

then the system is schedulable.

Note that if the right-hand side is given to the designer of component k then (s)he knows the supply of time available for component k and then (s)he can develop component k independently of other developers. Clearly, this is a formulation of composability.

When choosing dbfUB for each component, developers and systems integrators need to strike a balance: on the one hand, the right-hand side should be as detailed as possible in order to not increase pessimism; on the other hand, the right-hand side should not expose too much detail in order to facilitate system evolution. The technique in [4] can be used to achieve a trade-off with quantifiable cost in each of these aspects.

The technique described in this section only works if the jobs at run-time execute within what has been declared possible by the specified demand-bound function (or upper bound thereof). Therefore, we will now discuss policing.

3. RUN-TIME POLICING

Let us assume that each component has been specified with an upper bound on the demand-bound function (note that an important special case is when this upper bound is exact). Also, recall that $\text{dbfUB}(\tau^k, L)$ represents this upper bound for component k. It is clear from the definition of $\text{dbf}(\tau^k, L)$ that $\text{dbf}(\tau^k, L)$ is non-decreasing with increasing L. We will also assume that $\text{dbfUB}(\tau^k, L)$ is non-decreasing with increasing L. We also assume that $\forall L$, $\text{dbfUB}(\tau^k, L) \geq 0$. In addition, we assume that if a job has not finished by the time its deadline expires, then the job is killed.

Our goal is to design a run-time policing mechanism that ensures that jobs in a component do not violate the interface ($\text{dbfUB}(\tau^k, L)$) of the component. Note that this does not guarantee that all jobs in the component meet their deadlines. But using this policing mechanism ensures that jobs in a component do not violate its interface and this can be used (as seen in the previous section) to create a local schedulability test.

When discussing the policing mechanism, we start with a formulation of what the policing mechanism should achieve and then write simple pseudocode that does not perform exactly the way we want and then add new lines of code until it performs exactly the way we want.

Discussing a schedule. For a given schedule, let $\text{exec}(\tau^k, t_0, t_1)$ indicate the amount of execution performed in the time interval $[t_0, t_1]$ by jobs from τ^k with absolute deadlines at most t_1 and arrival time at least t_0. Then, we can express the property that the policing mechanism should ensure as fol-

lows: Make sure that for each time interval $[t_0,t_1]$, it holds that $\text{exec}(\tau^k,t_0,t_1) \leq \text{dbfUB}(\tau^k, t_1 - t_0)$. We call this the *Policing-correctness-property.*

We will now show that only a subset of those intervals $[t_0,t_1]$ needs to be checked. For a given schedule, suppose that the Policing-correctness-property is false. Then clearly, there exists a time interval $[t_0,t_1]$ such that

$$\text{exec}(\tau^k, t_0, t_1) > \text{dbfUB}(\tau^k, t_1 - t_0). \quad (7)$$

Recall that the beginning of Section 3 stated that we assume $\forall L$, $\text{dbfUB}(\tau^k, L) \geq 0$. Combining this with Eq 7 yields

$$\text{exec}(\tau^k, t_0, t_1) > 0. \quad (8)$$

Thus, there is at least one time in $[t_0,t_1]$ such that there is a job arriving at this time (because otherwise, $\text{exec}(\tau^k, t_0, t_1) = 0$ and it would contradict Eq 7). Also, there is at least one time in $[t_0,t_1]$ such that there is a job with absolute deadline expiring at this time (because otherwise, $\text{exec}(\tau^k, t_0, t_1) = 0$ and it would contradict Eq 8). Then we reason as follows: Let t_0' denote the smallest value such that $t_0 \leq t_0'$ and there is a job arriving at time t_0'. (From reasoning three sentences earlier, it can be seen that t_0' exists.) Clearly, since $t_0 \leq t_0'$ and since dbfUB is non-decreasing with respect to its 2nd parameter, it follows that

$$\text{dbfUB}(\tau^k, t_1 - t_0') \leq \text{dbfUB}(\tau^k, t_1 - t_0). \quad (9)$$

Also, from the definition of t_0', it follows that there is no job with arrival time in the time interval (t_0,t_0'). Hence, any execution in the time interval (t_0,t_0') is from jobs with arrival time t_0 or earlier. Let us consider two cases:

1. There is no job arriving at time t_0.

   From this condition, it follows that any execution in the time interval (t_0,t_0') is from jobs with arrival time strictly less than t_0. Hence, it follows that $\text{exec}(\tau^k,t_0',t_1) = \text{exec}(\tau^k,t_0,t_1)$.

2. There is a job arriving at time t_0.

   From this condition, it follows that $t_0' = t_0$. Hence, it follows that $\text{exec}(\tau^k,t_0',t_1) = \text{exec}(\tau^k,t_0,t_1)$.

Thus, regardless of the case, we have shown that

$$\text{exec}(\tau^k, t_0', t_1) = \text{exec}(\tau^k, t_0, t_1). \quad (10)$$

Combining Eq 7, Eq 9, and Eq 10 yields:

$$\text{exec}(\tau^k, t_0', t_1) > \text{dbfUB}(\tau^k, t_1 - t_0'). \quad (11)$$

That is, the time interval $[t_0',t_1]$ violates the Policing-correctness-property.

Let t_1' denote the largest value such that $t_1' \leq t_1$ and there is a job whose absolute deadline is at time t_1'. With analogous reasoning, we obtain that

$$\text{exec}(\tau^k, t_0', t_1') > \text{dbfUB}(\tau^k, t_1' - t_0'). \quad (12)$$

Hence, for a given schedule, if the Policing-correctness-property is violated for this schedule, then there exists a pair of time instants t_0' and t_1' such that t_0' is a time instant when there is a job arriving at time t_0' and t_1' is a time instant when there is a job whose absolute deadline expires at time t_1' and $\text{exec}(\tau^k,t_0',t_1') > \text{dbfUB}(\tau^k, t_1' - t_0')$.

Therefore, we can express the Policing-correctness-property in an equivalent form as follows: Make sure that for each time interval $[t_0,t_1]$ such that at time t_0 there is a job arriving and at time t_1 there is a job with absolute deadline expiring, it holds that $\text{exec}(\tau^k,t_0,t_1) \leq \text{dbfUB}(\tau^k, t_1 - t_0)$.

This expression is stated in terms of enumeration over time intervals. But we can express this equivalently as enumeration over pairs of jobs. Specifically, the Policing-correctness-property can be stated in an equivalent form as follows: Make sure that for each pair of jobs J' and J'' such that $A_{J''} + D_{J''} - A_{J'} \geq 0$, it holds that $\text{exec}(\tau^k,A_{J'},A_{J''} + D_{J''}) \leq \text{dbfUB}(\tau^k, A_{J''} + D_{J''} - A_{J'})$. It is straightforward to see that this is an equivalent formulation of Policing-correctness-property: $A_{J'}$ corresponds to t_0 and $A_{J''} + D_{J''}$ corresponds to t_1.

Run-time. Note that the Policing-correctness-property is evaluated over an entire schedule. We will now present a condition that can be used at run-time. Recall that Policing-correctness-property is about making sure that for each pair of jobs J' and J'' such that $A_{J''} + D_{J''} - A_{J'} \geq 0$, it holds that $\text{exec}(\tau^k,A_{J'},A_{J''} + D_{J''}) \leq \text{dbfUB}(\tau^k, A_{J''} + D_{J''} - A_{J'})$. Clearly, if this condition is false, then there exists a pair of jobs J' and J'' such that $A_{J''} + D_{J''} - A_{J'} \geq 0$, it holds that $\text{exec}(\tau^k,A_{J'},A_{J''} + D_{J''}) > \text{dbfUB}(\tau^k, A_{J''} + D_{J''} - A_{J'})$. Among those pairs, let us choose the pair J' and J'' such that $A_{J''} + D_{J''}$ is the smallest. Thus, there is no pair of jobs where the 2nd job has earlier absolute deadline that violates the Policing-correctness-property. Also, since $\text{exec}(\tau^k,A_{J'},A_{J''} + D_{J''}) > \text{dbfUB}(\tau^k, A_{J''} + D_{J''} - A_{J'})$ it follows that there exists an x (with $x \in [0,A_{J''} + D_{J''} - A_{J'}]$) such that $\text{exec}(\tau^k,A_{J'},A_{J'}+x) = \text{dbfUB}(\tau^k, A_{J''} + D_{J''} - A_{J'})$ and just after time $A_{J'}+x$, there is execution and Policing-correctness-property is violated. Let us choose the smallest such x. Let $A_{J'}+y$ denote the latest scheduling event before $A_{J'}+x$ (a scheduling event is a time when a job arrives or a deadline expires or a context switch occurs). Intuitively, $A_{J'}+x$ is the time when Policing-correctness-property is violated and $A_{J'}+y$ is the latest time when Policing-correctness-property is not violated and we can do something about it. Clearly, for y it holds that

$$\text{exec}(\tau^k,A_{J'},A_{J'}+y) \leq \text{dbfUB}(\tau^k, A_{J''} + D_{J''} - A_{J'}). \quad (13)$$

Thus, let us define δ as $\delta = \text{dbfUB}(\tau^k, A_{J''} + D_{J''} - A_{J'}) - \text{exec}(\tau^k,A_{J'},A_{J'}+y)$. It is easy to see (from Eq 13) that $\delta \geq 0$. Intuitively, δ is the amount of margin that we have at time $A_{J'}+y$, i.e., δ is the amount of execution that we can allow after $A_{J'}+y$ without violating Policing-correctness-property.

Pseudocode. From the above discussion, we obtain that policing can be achieved as follows: Whenever a scheduling event (job arrival, deadline expiring or a context switch) occurs, do the following:

1. Let timeb denote the current time.
2. Find the largest dbfslack such that for each pair of jobs J' and J'' such that the arrival time of J' is at most timeb and the arrival time of J'' is at most timeb

and $A_{J''} + D_{J''} - A_{J'} \geq 0$, it holds that dbfslack + $\text{exec}(\tau^k, A_{J'}, A_{J''} + D_{J''}) \leq \text{dbfUB}(\tau^k, A_{J''} + D_{J''} - A_{J'})$.

3. Set up a timer to expire at time timeb + dbfslack (i.e., dbfslack time units in the future) and when the timer expires, treat it as a scheduling event.

The reason why this works is that in Step 2., we compute dbfslack and it can be seen that regardless of the schedule generated during the next dbfslack time units, it holds that if no new job arrives, then the interface will not be violated during the next dbfslack time units. Let us now discuss how this can be implemented with pseudocode.

The accounting of execution of component k can be performed as follows:

```
at initialization do
  IS := empty set
  JS := empty set

at each context switch or time when a job
arrives or time when a deadline expires do
  let prev denote the task that executed
    before this event
  let next denote the task that executes
    after this event
  timeb := read current time
  if current context switch is a job arrival
  then
    create a new object J'''
    J'''.rA := timeb
    J'''.rd := timeb +
      deadline of the task next
    JS := JS union J'''
  end if
  if prev != null then
    create a new object I
    I.rstart := timea
    I.rstop := timeb
    IS := IS union I
  end if
  timea := timeb
```

The code above performs accounting. It records job arrivals and deadlines, and stores them in sets. (In the pseudocode above: JS means Job-Set; rA means recorded-arrival-time; rd means recorded-absolute-deadline.) It also records the time intervals when the component executes. (In the pseudocode above: IS means Interval-Set; ristart means recorded-interval-start-time; ristop means recorded-interval-stop-time.) We would like to add policing to it. This can be achieved by considering each job arrival time and each deadline and for this time interval, consider the amount of execution that the component has performed and see how much extra execution can be allowed; and then take the minimum of that. Let dbfslack denote this value. We set up a timer interrupt dbfslack time units in the future. Hence, policing of component k can be performed as follows:

```
at initialization do
  IS := empty set
  JS := empty set

at each context switch or time when a job
arrives or time when a deadline expires do
  let prev denote the task that executed
    before this event
  let next denote the task that executes
    after this event
  timeb := read current time
  if current context switch is a job arrival
  then
    create a new object J'''
    J'''.rA := timeb
    J'''.rd := timeb +
      deadline of the task next
    JS := JS union J'''
  end if
  if prev != null then
    create a new object I
    I.rstart := timea
    I.rstop := timeb
    IS := IS union I
  end if
  dbfslack := infinity
  for each J' in JS do
    for each J'' in JS do
      sumexec := 0
      for each I in IS do
        if (J'.rA <= I.ristart) and
          (I.ristop <= J''.rd) then
            sumexec := sumexec +
              I.ristop - I.ristart
        end if
      end for
      delta :=
        dbfUB(τ^k, J''.rd - J'.rA)
         - sumexec
      if delta<dbfslack then
        dbfslack := delta
      end if
    end for
  end for
  set up a timer to expire
    at time timeb+dbfslack;
    when it expires, treat it
    as a context switch.
  timea := timeb
```

There are four errors in the code above. First, when computing delta, we consider time intervals of duration J".rd - J'.rA. If this is negative, then we will evaluate dbfUB for a negative value and this is not defined. Hence, we should check for this in the double for-loop. Second, in the double for-loop, we should only count execution from jobs with absolute deadline at most J".rd and arrival time at least J'.rA. Third, if dbfslack is negative then it is not obvious what setup timer would do. Fourth, if dbfslack is positive but very small (e.g., 1 microsecond) then this time may have elapsed even before the timer interrupt has been set up. We introduce a parameter THRESHOLD which is used for this decision. Its value depends on the computer platform. In an ideal platform THRESHOLD = 0; but as a ballpark estimate, a reason-

able assignment is THRESHOLD = 1 microsecond. Hence, policing of component k can be performed as follows:

```
at initialization do
  IS := empty set
  JS := empty set

at each context switch or time when a job
arrives or time when a deadline expires do
  let prev denote the task that executed
    before this event
  let next denote the task that executes
    after this event
  timeb := read current time
  if current context switch is a job arrival
  then
    create a new object J'''
    J'''.rA := timeb
    J'''.rd := timeb +
      deadline of the task next
    JS := JS union J'''
  end if
  if prev != null then
    create a new object I
    I.rstart := timea
    I.rstop := timeb
    I.rA := arrival time
      of the job of task prev
    I.rd := absolute deadline
      of the job of task prev
    IS := IS union I
  end if
  dbfslack := infinity
  for each J' in JS do
    for each J'' in JS do
      if J''.rd - J'.rA>0 then
        sumexec := 0
        for each I in IS do
          if (J'.rA <= I.ristart) and
             (I.ristop <= J''.rd) and
             (J'.rA <= I.rA) and
             (I.rd <= J''.rd) then
              sumexec := sumexec +
                I.ristop - I.ristart
          end if
        end for
        delta :=
          dbfUB(τ^k, J''.rd - J'.rA)
           - sumexec
        if delta<dbfslack then
          dbfslack := delta
        end if
      end if
    end for
  end for
  if dbfslack > THRESHOLD then
    set up a timer to expire
      at time timeb+dbfslack;
      when it expires, treat it
      as a context switch.
  else
    stop all execution in this
      component and do not allow
      any future execution
      in this component
  end if
  timea := timeb
```

4. WHY IS MEMORY-EFFICIENT POLICING HARD?

We have seen a policing mechanism in the previous section. It kept track of all jobs and all execution segments in the schedule. Doing so is undesirable because for long-lived systems, this run-time dispatching consumes lots of memory and its run-time overhead may be large. One may wonder if it is possible to create a policing mechanism that can forget; i.e., if it is possible to create a policing mechanism that does not depend on data that is sufficiently old. In this section, we show why creating such a mechanism is difficult.

Consider component k and assume that there is one task in this component and that this task is an implicit-deadline sporadic task with the following characteristics: $T_1^k = 1, D_1^k = 1, C_1^k = 1/\text{INT}$. Here INT is a positive integer greater than or equal to 2. Also, assume that the interface is the following: $\text{dbfUB}(\tau^k, L) = 0$ if $L < 1$; otherwise $\text{dbfUB}(\tau^k, L) = 1$. And assume that the policing mechanism is the one in the previous section configured with THRESHOLD = 0.

Consider the case that a job of τ_1 arrives at time 0 and jobs of this task arrive periodically. Then, at time 0, the policing code will execute and it will compute dbfslack = 1. The 1st job of τ_1 will execute during the time interval $[0, 1/\text{INT})$. Then the processor is idle during the time interval $[1/\text{INT}, 1)$. Then at time 1, a job of τ_1 arrives and it will be allowed to execute immediately when it arrives and it finishes at time $1 + 1/\text{INT}$. This behavior keeps repeating until time $\text{INT} - 1$. At this time, a job of τ_1 arrives. The policing will allow it to execute and it will finish at time $\text{INT} - 1 + 1/\text{INT}$. Then at time INT, a new job of τ_1 arrives and the policing mechanism is invoked. The policing mechanism considers many time intervals; one of them is the arrival of the 1st job until the deadline of the INT + 1:th job (which is at time INT + 1). There is so far $\text{INT} \times 1/\text{INT}$ units of execution performed in this time interval of jobs with arrival time greater than or equal to the beginning of the time interval and absolute deadline less than or equal to the end of the time interval. Also, note that for this time interval $\text{dbfUB}(\tau^k, L) = 1$. Hence we compute $\text{dbfslack} = 1 - \text{INT} \times 1/\text{INT}$ and this yields dbfslack = 0. Since dbfslack = 0 at time INT, the run-time policing mechanism suspends the component at time INT. This is the correct behavior. Note, however, that the INT−2 first jobs have deadlines before current time and they still impact the result; if their execution would have not been considered then we would have obtained the wrong result. We can repeat this reason for any positive INT. By choosing an infinite INT, we obtain that when performing policing, it may be necessary to consider a time interval that started at an arrival time infinitely number of jobs in the past and when counting execution, it may be necessary to consider such execution infinitely in the past.

5. RELATED WORK

The literature on hierarchical scheduling, composability, and compositionality is vast; here we only survey some of the previous work. When rate-monotonic scheduling was developed as a comprehensive framework, it was recognized that many systems have software whose resource consumption is hard to characterize; e.g., it is hard to find its worst-case execution time or minimum inter-arrival time. For this reason, reservation-based frameworks were developed (see for example [13]). The run-time behavior of such a framework is as follows: A task may be associated with a server task and this server task is scheduled as a normal task (for example with an execution time, often called budget, and a period) and if a task is associated with a server task then it is only allowed to execute when the server task executes. In this way, if a task τ_i is in a server task and if τ_i experiences an execution overrun or arrives more often than expected then its impact on other tasks is bounded and it is bounded by the parameters of the server task. Later works created such reservation for EDF; one example of that is the constant-bandwidth server (CBS) [1].

Researchers realized that reservation-based frameworks can be used to form hierarchical scheduling; some work that did so with EDF include [14, 9]. With the focus on hierarchical scheduling, Feng and Mok developed [11] a resource model which states that a root scheduler supplies, in a time interval of duration t, at least $(t-\Delta^k)*\alpha^k$ units of execution to component k. Shin and Lee developed [15] another model where a component k is characterized by its period and execution time and with this, presented a so-called supply-bound function; this work could be used for components that use fixed-priority as a local scheduler or components that use EDF as a local scheduler. Later works have focused on resource sharing, specifically the question, if a task τ_{lock} executes within a critical section and its current budget has reached zero, what should the scheduler do? There may be other tasks that will request this critical section and these requests can be granted only if the task that currently holds that critical section (τ_{lock}) has released it. In order for the task (τ_{lock}) to release the critical section, it must be able to execute and in order for this task (τ_{lock}) to execute, it must have a current budget greater than zero. Different solutions for this have been developed; see for example [8, 7, 6]. Clearly, server parameters must be selected in order to ensure schedulability; this has been the focus of [2].

Recall that early in this paper, we pointed out that bandwidth-like schemes can suffer from poor performance and we have suggested that using the demand-bound function is an interface that does not suffer from this drawback. A similar point has been made in [12, 10]; they also discuss approximate policing of such an interface.

6. CONCLUSIONS

This paper has presented ideas for how an interface originally developed for compositionality can be used to achieve composability. We have also seen a design of a policing mechanism. We left open the question on how to perform policing with low memory consumption and low CPU overhead. The policing presented here makes sure that the interface is not violated but the policing does not depend on any scheduler used. Naturally, all of our results apply to EDF but they also apply to Least-Laxity-First (LLF). For future work on policing, it may be worth assuming that EDF is used and exploit this to reduce memory and CPU overhead of the policing mechanism.

Acknowledgment

Copyright 2016 ACM This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. [Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM-0003263

7. REFERENCES

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *RTSS*, 1988.

[2] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *EMSOFT*, 2004.

[3] B. Andersson. A pseudo-medium-wide 8-competitive interface for two-level compositional real-time scheduling of constrained-deadline sporadic tasks on a uniprocessor. In *CRTS*, 2009.

[4] B. Andersson. A preliminary idea for an 8-competitive, log2 DMAX + log2 log2 (1/U) asymptotic-space, interface generation algorithm for two-level hierarchical scheduling of constrained-deadline sporadic tasks on a uniprocessor. In *CRTS*, 2010.

[5] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. In *Real-Time Systems*, pages 301–324, 1990.

[6] M. Behnam, T. Nolte, M. Asberg, and R. J. Bril. Overrun and skipping in hierarchically scheduled real-time systems. In *RTCSA*, 2009.

[7] M. Behnam, I. Shin, T. Nolte, and M. Nolin. SIRAP: a synchronization protocol for hierarchical resource sharing in real-time open systems. In *EMSOFT*, 2007.

[8] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *RTSS*, 2006.

[9] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *RTSS*, 1997.

[10] F. Dewan and N. Fisher. Efficient admission control for enforcing arbitrary real-time demand-curve interfaces. In *RTSS*, 2012.

[11] X. Feng and A. K. Mok. A model of hierarchical real-time virtual resources. In *RTSS*, 2002.

[12] P. Kumar, J.-J. Chen, and L. Thiele. Demand bound server: Generalized resource reservation for hard real-time systems. In *EMSOFT*, 2011.

[13] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *RTSS*, 1987.

[14] G. Lipari and S. K. Baruah. A hierarchical extension to the constant bandwidth server framework. In *RTAS*, 2001.

[15] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *RTSS*, 2003.